Vol.2014-SE-186 No.2
2014/11/13

開発履歴分析を用いたコードクローン内外における欠陥発生率の調査

中山 直輝[1,a)]　吉田 則裕[2,b)]　藤原 賢二[1,c)]　飯田 元[1,d)]

概要：コードクローンとは，ソースコード中の互いに一致または類似したコード片を指し，主に開発者が行うコピーアンドペーストによって発生する．近年，コードクローン解析に基づくソフトウェアの欠陥検出手法が提案されており，実際に欠陥を検出した事例も報告されている．コードクローンに着目して欠陥を探す際に，欠陥がコードクローン内に含まれる場合もあれば，コードクローン外に存在する場合もあるため，開発者はコードクローン内外の両方について欠陥の有無を調べなければならない．しかし既存研究では，コードクローン内外のどちらで欠陥が多く発生し，開発者がどちらを優先的に検査すべきかを示す指標は得られていない．そこで本研究では，オープンソースソフトウェアを対象としたコードクローン内外における欠陥発生率の調査を行った．調査の結果，コードクローン内の欠陥発生率よりもコードクローン外の欠陥発生率の方が高いことが分かった．また，コードクローンとの距離に基づき，コードクローン外で特に欠陥発生率が高い範囲を調査した結果，コードクローンの上側では，コードクローンから遠ざかるにつれて欠陥発生率が低下することが分かった．

キーワード：コードクローン，リポジトリマイニング，欠陥管理システム，ソフトウェア保守

Investigation of Defect Rate within Code Clones and the Surrounding Code Fragments Using Repository Mining Technique

Naoki Nakayama[1,a)]　Norihiro Yoshida[2,b)]　Kenji Fujiwara[1,c)]　Hajimu Iida[1,d)]

Abstract: Code clone is a duplicate code fragment in the source code of software and generated by developer's copy and paste. Recently, some researchers proposed defect detection method based on code clone analysis and the proposed methods were able to detect defects in certain projects. Detecting defect based on code clone analysis may require developer to spend effort to check whether there is any defect because defect may exist within the code clone or the surrounding code fragments. However, there is no existing research clarifying which area, within code clone or the surrounding code fragments, poses higher defect rate. Therefore, there is no standard for developers to follow for prioritizing the order of inspection. This research investigated the defect rate within code clone and the surrounding code fragments on three open source softwares. The investigation revealed that the defect rate of surrounding code fragments is higher than the one of code clone. Furthermore, by measuring the distance from code clone to surrounding code fragments, we investigated the area that has higher defect rate within surrounding code fragments. According to the investigation, the defect rate of surrounding code fragments that are above the code clone is decreasing as the distance gets further.

Keywords: Code Clone，Repository Mining，Bug Tracking System，Software Maintenance

[1] 奈良先端科学技術大学院大学
Nara Institute of Science and Technology
[2] 名古屋大学 Nagoya University
[a)] nakayama.naoki.my7@is.naist.jp
[b)] yoshida@ertl.jp
[c)] kenji-f@is.naist.jp
[d)] iida@itc.naist.jp

1. はじめに

コードクローンとは，ソースコード中に存在する互いに一致または類似したコード片を指し，主に開発者が行うコピーアンドペーストによって発生する [1]．コードクローンの存在はソフトウェアの品質に悪影響を及ぼすと言われている [2]．例えば，開発者が 1 つのコード片を編集した場合，そのコード片のコードクローンに対しても同様の編集作業が必要となる可能性があり，開発者は全てのコードクローンに対して編集作業の是非を検討しなければならなくなり，ソフトウェアの保守性を低下させてしまう [3]．また，全てのコードクローンに対して一貫した編集が必要であるにもかかわらず，開発者が編集作業を怠ると，編集されなかったコード片が原因で欠陥が発生する恐れがある．

このような背景を受け，近年，コードクローン解析に基づいてコードクローンに起因する欠陥を検出する手法が提案されている．Jiang らは，コードクローンを内包する処理構造の不一致に着目する欠陥検出手法を提案している [4]．**図 1** は Jiang らの手法によって Linux カーネルから実際に検出された欠陥の例である．sample1.c（簡単のため，ファイル名や変数名等を変更している）の 18 行目から 29 行目と sample2.c の 63 行目から 74 行目は互いにコードクローン関係であり，それぞれのコード片をコードクローン 1 およびコードクローン 2 とすると，コードクローン 1 を内包する if 文（以降，if 文 1 とする）とコードクローン 2 を内包する if 文（以降，if 文 2 とする）の条件式に差異が見られる．具体的には，if 文 1 と if 文 2 の条件式内で呼び出している関数が異なっており，この不一致に着目してコードクローン 2 に含まれる欠陥を検出している．コードクローン 2 に含まれる欠陥とは，本来 if 文 2 で呼び出している関数 strcmp をコードクローン 2 の中でも呼び出さなければならないが，不適切な関数 strncmp を呼び出してしまい生じた論理エラーのことである．この欠陥は，開発者がコピーアンドペーストを用いてコード片を再利用した後，コピーしたコード片（コードクローン 2）をコピー先のプログラムに合わせて適切に修正しなかったことが原因で生じたと考えられる．

図 1 で示した例では欠陥がコードクローン内に存在したが，コードクローンに起因する欠陥がコードクローン外で発生する場合もあり，**図 2** がその例である．sample3.c の 1029 行目から 1035 行目と sample4.c の 1321 行目から 1327 行目が互いにコードクローン関係であり，それぞれのコード片をコードクローン 3 およびコードクローン 4 とすると，コードクローン 3 内では関数 function1 が，コードクローン 4 内では関数 function2 が呼び出されている．この例では，コピーアンドペーストによって発生したコードクローンを開発者が適切に修正しており，図 1 のような

```
File: sample1.c
17: if(len >= 9 && strncmp(buf, "EESOX", 9) == 0){
18:     buf += 9;
19:     len += 9;
20:
21:     if(len >= 5 && strncmp(buf, "term=", 5) == 0){
            ......
28:     } else
29:         ret = -EINVAL;
30: } else
31:     ret = -EINVAL;

File: sample2.c
62: if(len >= 9 && strcmp(buf, "CUMANA") == 0){
63:     buf += 9;
64:     len += 9;
65:
66:     if(len >= 5 && strncmp(buf, "term=", 5) == 0){
            ......
73:     } else
74:         ret = -EINVAL;
75: } else
76:     ret = -EINVAL;
```

図 1 コードクローン内に欠陥が発生した例
Fig. 1 Example of defect within code clone.

間違った関数を呼び出すといった欠陥は発生していない．しかし，開発者が修正した関数呼び出しに関連する欠陥がコードクローンの外（sample4.c の 1315 行目）で発生してしまっている．具体的には，sample3.c および sample 4 .c が属するプロジェクトには制約（関数 function1 を呼び出す前には関数 CheckStatusA を，関数 function2 を呼び出す前には関数 CheckStatusB を呼び出す）が存在し，問題のコード片はこの制約に違反している．本来，開発者はコードクローン 4 内で呼び出す関数を function 1 から function2 に修正する際に，sample4.c の 1315 行目で CheckStatusB を呼び出すように修正しなければならなかったが，その作業を怠ったことで欠陥が発生している．このように，コード片のコピーアンドペーストを行うと，コピー先のプログラムに欠陥を発生させてしまう恐れがある．

このように，コードクローンに起因する欠陥はコードクローンの内外に存在する．そのため，開発者がコードクローンに着目してソースコード中の欠陥を探す場合は，コードクローンの内外それぞれについて欠陥の有無を確認する必要がある．しかし，コードクローンと欠陥の関係を調査した既存研究では，コードクローン内外のどちらで欠陥が多く発生し，開発者がどちらを優先的に検査すべきかを示す指標は得られていない．そこで本研究では，オープンソースソフトウェア（以下，OSS とする）においてコードクローン内外のどちらで欠陥が多く発生しているかを定量的に調査する．そして，コードクローン内よりもコードクローン外の方に欠陥が多く発生している場合，コードク

```
File: sample3.c
1023: CheckStatusA();
      ......
1029: if( ... )
1030: {
1031:     ...
1032:     function1();
1033:     ...
1034: }else
1035:     ...

File: sample4.c
1315: CheckStatusA();
      ......
1321: if( ... )
1322: {
1323:     ...
1324:     function2();
1325:     ...
1326: }else
1327:     ...
```

図 2　コードクローン外に欠陥が発生した例
Fig. 2　Example of defect within the surrounding code fragments.

ローン外の特に欠陥が多く発生している場所をコードクローンとの距離に基いて調査する．コードクローンとの距離とは，ソースコード中で対象のコード片がコードクローンから何行離れているかを表す指標である．例えば，コードクローンの上側 100 行以内で特に欠陥が多く発生していた場合，開発者はコードクローンの上側 100 行以内を優先的に検査すると効率的な欠陥探索を行うことができると考えられる．

以降，2 章では本研究の関連研究を紹介し，3 章では本研究の調査手法について述べる．4 章では調査結果を示し，5 章では調査結果に関する考察を述べる，最後に，6 章で今後の課題に触れながら本稿をまとめる．

2. 関連研究

これまでに，さまざまな研究者がコードクローンと欠陥の関係を定量的に調査している．本章では，その中から本研究に関連する 2 つの既存研究を紹介する．

Rahman らは，C で開発された OSS において，ソースコード全体におけるコードクローンの割合と欠陥におけるコードクローンの割合を比較し，コードクローンが欠陥を発生させる割合を調査している [5][6]．具体的には，OSS の開発履歴と欠陥管理システムの情報に基づき，対象の OSS プロジェクトで実際に報告された欠陥を含むソースファイルと欠陥に該当するコード片の行番号を特定し，既存のコードクローン検出ツールによって得られるソースコード中のコードクローンの位置情報と照らし合わせることで，ソースコード全体におけるコードクローンの割合と欠陥におけるコードクローンの割合を計算して比較している．本研究で行う調査もこの手法に基づいているため，より詳細な説明は 3 章で述べる．Rahman らの調査の結果，欠陥におけるコードクローンの割合がソースコード全体におけるコードクローンの割合よりも低いことが分かった．つまり，コードクローンが原因で欠陥が発生し，ソフトウェアの品質に悪影響が及ぶという従来の知見とは異なる結果が得られた．

Sajnani らは，Java で開発された OSS のソースコードを対象に，コードクローンに該当するコード片とコードクローンでないコード片における欠陥の割合を比較し，コードクローン内外のどちらで欠陥が多く発生しているかを調査している [7]．この調査では，Java のソースコード中でエラーとなる可能性が高いコード片を検出する FindBugs[8] という欠陥検出ツールを用いてソースコード中の欠陥位置を特定している．この調査の結果，コードクローン外における欠陥の割合がコードクローン内における欠陥の割合よりも 3.7 倍高いことが確認された．

以上が関連研究である．Rahman らの調査ではソースコード中でコードクローンが特に欠陥を発生させる原因ではないことを明らかにしているが，コードクローン内外における欠陥の割合を直接比較していない点で，本研究とは調査内容が異なっている．また，Sajnani らの調査では，コードクローン内外における欠陥の割合を直接比較している一方で，欠陥検出ツールが誤検出を起こす可能性が高いといった問題点が存在する．FindBugs は，あくまでエラーとなる可能性が高いコード片を検出するツールであることから，欠陥でないコード片を誤って欠陥として検出してしまう確率が最大で 50%に及ぶことが Hovemeyer らの調査で分かっている [9]．そこで本研究では，Rahman らの調査と同様に，OSS の欠陥管理システムで報告された欠陥情報を用いてコードクローン内外における欠陥の割合を調査する．

3. 調査手法

本調査における 2 つの調査項目を以下に示す．

調査 1
コードクローン内外のどちらで欠陥が多く発生しているか．

調査 2
コードクローン外において特に欠陥が多く発生している範囲はどこか．

図 3 は調査手法の概要である．前述のとおり，本研究で行う調査の手法は Rahman らの調査手法に基づいている．手順は大きく 3 つのステップに分かれており，以降 3.1 では欠陥位置の特定方法について，3.2 ではコードクローンの検出方法について，3.3 では各調査項目における欠陥発

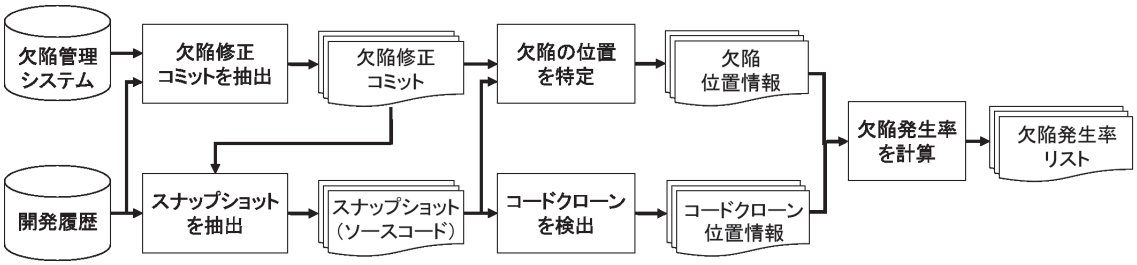

図 3　調査手法の概要
Fig. 3　An overview of investigation method.

生率の計算方法について述べる．

3.1　欠陥位置の特定

OSS の開発履歴と欠陥管理システムの情報を用いて欠陥の位置情報（欠陥が存在するファイルと欠陥に該当するコード片の行番号）を特定する．本研究では，版管理システムとして Git[*1] を，欠陥管理システムとして Bugzilla[*2] を採用しているプロジェクトを対象に調査する．

まず，対象プロジェクトの Git リポジトリで管理されている全ての変更履歴（以降，コミットとする）からコミットログを取得する．次に，取得したコミットログを Śliwerski らの szz アルゴリズム [10] に基いて解析し，全コミットから欠陥修正コミットである可能性が高いコミットだけを抽出する．具体的には，開発者がコミットログに残した文章から ‘Bug’，‘Defect’，‘Fixed’ のような欠陥修正に関する文字列と ‘#123456’ といった Bugzilla の欠陥情報が持つ固有の ID（以降，欠陥 ID とする）を表す数字列を抽出し，これらのキーワードの数や重要度に基いて予め定義したレベルが閾値以上となるコミットのみを欠陥修正コミットとして抽出する．なお，コミットログから抽出した欠陥 ID が欠陥管理システムに実在しない場合および対象の欠陥の深刻さが ‘enhancement’ である場合，そのコミットは除外する．Bugzilla では，欠陥の深刻さ（Severity）を 7 段階で表現しているが，‘enhancement’ は開発者に対して機能の追加や改善を依頼する際に用いられるものであるため，欠陥とは直接関係がないと考え除外する．さらに，欠陥修正コミットでないコミットを誤って抽出してしまうことを避ける（False Positive を減らす）ため，抽出した欠陥修正コミットに対して Bachmann らのフィルタリング手法 [11] を適用する．このフィルタリング手法では，版管理システムにおけるコミット日時および欠陥管理システムに登録された欠陥修正日時の差に着目し，版管理システムにコミットされる前後 7 日以内に欠陥管理システムで欠陥修正報告が行われたコミットのみを欠陥修正コミットと見なし，その他のコミットは除外する．

以上の手順で Git リポジトリから抽出した欠陥修正コミットについて，コミット直後のリビジョンを r とすると，修正された欠陥はコミット直前のリビジョン $r-1$ に存在する．したがって，欠陥修正コミットにおいて修正された $r-1$ に存在するコード片が欠陥に該当する．$r-1$ における欠陥の行番号は UNIX の Diff コマンドを用いて特定する．r と $r-1$ の間で修正された全てのファイルに対して Diff コマンドを適用し，変更または削除された行を欠陥と見なす．$r-1$ から r の間で新しく追加された行については，欠陥の直接的な原因でないと判断して欠陥から除外する．

3.2　コードクローンの検出

コードクローンの検出は Jiang らの検出ツールである DECKARD を用いる [12]．DECKARD は，入力として与えられたソースコードを抽象構文木に変換し，抽象構文木上の互いに類似する部分木をコードクローンとして検出する．ツールの主な出力はコードクローンの位置情報（コードクローンが存在するファイルのファイルパスおよびコードクローンに該当するコード片の行番号などの情報）である．DECKARD に与えるパラメータ設定は，Rahman らの手法に従って ‘minimum token’ は 50，‘similarity’ は 1.0，‘stride’ は 2 とする．本研究でコードクローン検出の対象となるソースコードは，3.1 で述べた欠陥が存在するリビジョンのスナップショットである．しかし，欠陥が存在する全てのスナップショットに対してコードクローン検出を行うと計算時間が膨大となってしまうため，月初めのリビジョンのスナップショット（以降，ステージングスナップショットとする）に限定してコードクローンを検出する．具体的には，3.1 で特定した欠陥を含むリビジョンのスナップショットがステージングスナップショットでなかった場合，そのリビジョンがコミットされた月のステージングスナップショットに着目し，欠陥がステージングスナップショット中のどのコード片に該当するかを UNIX の Diff コマンドを用いて特定する．**図 4** はその例である．例えば，ある月の欠陥修正直後のリビジョンを r，欠陥が存在するリビジョンを $r-1$，そのステージングスナップショットを ss，さらに $r-1$ と ss における欠陥の行をそれぞれ l_{r-1} と l_{ss} とすると，まず ss から $r-1$ への変更点を

[*1] http://git-scm.com/
[*2] http://www.bugzilla.org/

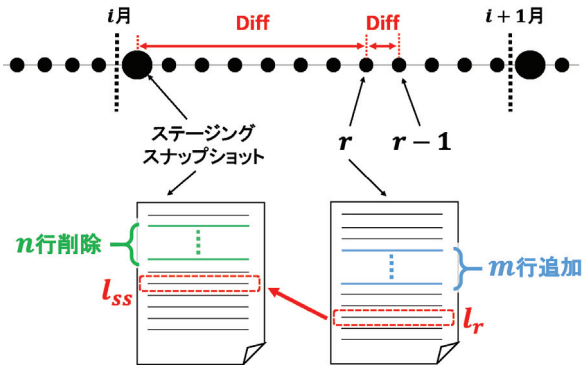

図 4　欠陥の位置を特定する方法

Fig. 4　Process of specifying the line number of defect at its staging snapshot.

Diff コマンドによって求め，もし l_{r-1} より上側のコード片において m 行の追加と n 行の削除が行われていれば，l_{ss} は $l_{r-1}-m+n$ となる．l_{r-1} が ss で存在しない場合（l_{r-1} が ss から $r-1$ の間で新しく追加された行であった場合），その行は欠陥から除外する．

3.3　欠陥発生率の計算と比較

本研究では，調査 1 と調査 2 の両調査において**欠陥発生率**の比較を行う．欠陥発生率とは，あるコード片において欠陥が含まれる割合である．例えば，100 行のコード片のうち 10 行が欠陥であった場合の欠陥発生率は 0.1 である．以降では，それぞれの調査項目における欠陥発生率の計算方法および比較方法について述べる．2 つの調査の共通事項として，本研究では前節で述べたステージングスナップショットのうち欠陥とコードクローンの両方が存在するファイルのみを調査対象とする．なぜなら，本研究は開発者がプロジェクト中から欠陥の存在が疑われるファイルを絞り込んだ上で，コードクローンに着目して欠陥を探す作業を想定しているからである．例えば，これまで動作していたプログラムが正常に動作しなくなり，前回実行時から編集されたファイルが特定されている状態で，それらのファイルにコードクローンが多く確認されたため，開発者がコードクローンに着目して欠陥を探す場面が挙げられる．

調査 1 では，調査対象の全てのファイルにおいてコードクローン内外それぞれの欠陥発生率を計算する．例えば，あるソースファイルのうちコードクローンであるコード片が合計で 500 行あり，そのうち欠陥であるコード片が 10 行であったとすると，コードクローン内の欠陥発生率は 0.02 となる．コードクローン外についても同様に計算し，対象の全てのファイルにおいてコードクローン内外の欠陥発生率をそれぞれ算出する．そして，コードクローン内の欠陥発生率とコードクローン外の欠陥発生率の中央値を比較し，それぞれの中央値に差があるかどうかを統計的に検定する．

調査 2 では，まず調査対象のファイルに存在する全てのコードクローンについて，コードクローン外における欠陥の有無をコードクローンとの距離毎に確認する．コードクローンとの距離とは，コードクローンから何行離れているかを表す指標であり，コードクローンの上側と下側は別々に扱う．もしコードクローンの上側および下側にソースコードが存在しない場合，そのコードクローンは調査から除外する．そして，欠陥の有無に基づき，コードクローン外における欠陥発生率をコードクローンとの距離毎に算出する．例えばコードクローンの上側 1 行目に着目すると，コードクローンの上側 1 行目にソースコードが存在するコードクローンの数（サンプル数）が 1000 であり，コードクローンの上側 1 行目に欠陥が存在するコードクローンの数が 10 であった場合，その欠陥発生率は 0.01 となる．この計算をコードクローンの上側と下側のそれぞれについて行う．コードクローンとの距離が大きくなると，コードクローン外にソースコードが存在しないコードクローンが増え，サンプル数が減少すると考えられる．そこで，サンプル数が全コードクローン数の 95%より小さくなった時点で調査を終了する．

4.　調査結果

本章では，C で開発された 3 つの OSS（gimp, evolution, nautilus）に対して行った調査の結果を示す．**表 1** は 3 つのプロジェクトの詳細である．ステージングスナップショット数は本研究で対象としたステージングスナップショットの数を表しており，平均 LOC は対象のステージングスナップショット毎の LOC の平均値を，対象ファイル数は本研究で対象にしたファイル（欠陥とコードクローンの両方が存在するファイル）の総数を表している．

表 1　対象プロジェクトの概要
Table 1　Summary of studied projects.

	総スナップショット数	総 LOC	総ファイル数
gimp	82	851708	1018
evolution	41	721344	792
nautilus	60	251260	191

4.1　調査 1：コードクローン内外のどちらで欠陥が多く発生しているか．

図 5 から**図 7** は，コードクローン内外におけるファイル毎の欠陥発生率の分布を表した箱ひげ図であり，縦軸は欠陥発生率を表す．比較の結果，全てのプロジェクトにおいてコードクローン外の中央値がコードクローン内の中央値に比べて高かった．**表 2** に各プロジェクトの中央値およびコードクローン内外の差の検定結果を示す．コードクローン内外の差の検定には Wilcoxon の順位和検定を用いた．

なぜなら，対象のデータに対して Kolmogorov-Smimov 検定（以降，KS 検定とする）を行った結果，全データにおいて正規性が認められなかったからである．Wilcoxon の順位和検定の結果，evolution と nautilus では有意水準 5%で有意差が確認された．つまり，evolution および nautilus ではコードクローン内よりコードクローン外の方が欠陥発生率が高いと言える．

4.2 調査 2：コードクローン外において特に欠陥が多く発生している範囲はどこか．

コードクローン外における欠陥発生率の推移を**図 8** に示す．各図の横軸はコードクローンとの距離であり，コードクローンから何行離れているかを表す．青色の折れ線グラフはコードクローンの上側を，赤色の折れ線グラフはコードクローンの下側を表している．コードクローンの下側のグラフが途中で切れているのは，コードクローンの上側に比べてコードクローンの下側ではコード片が存在せずに除外されるコードクローンが多く，サンプル数が先に全体の 95%を下回ったからである．ここで，コードクローンから遠ざかる（距離が大きくなる）につれて欠陥発生率がどのように推移しているかを調べるため，データをコードクローンとの距離が近い範囲と遠い範囲の 2 つに分割し，両者の平均値に差が見られるかを検定した．検定には t 検定（データが不等分散の場合は Welch の t 検定）を用いた．検定の結果を**表 3** から**表 6** に示す．t 検定を用いた理由は，KS 検定でデータに正規性が認められたからである．範囲の分割方法として，近い範囲と遠い範囲のデータ数がそれぞれ 1 対 1 になるように分割する方法と 1 対 2 になるように分割する方法の 2 つのパターンで調査した．検定はコードクローンの上側と下側のそれぞれで行い，結果を表 3 から表 6 に示す．表 3 と**表 4** より，gimp と nautilus では，コードクローンの上側における欠陥発生率の平均値が，コードクローンから遠い範囲よりも近い範囲で高く，有意水準 5%で有意差が見られた．つまり，コードクローンから遠ざかると欠陥発生率が低下していると言える．一方，コードクローンの下側については，コードクローンから近い範囲と遠い範囲で有意差は見られなかった．また，t 検定（データが不等分散の場合は Welch の t 検定）を用いてコードクローンの上側と下側で欠陥発生率に差が見られるかを調べた．t 検定を用いた理由は前述のとおりである．コードクローンの上側と下側でデータ数が異なるため，少ない方にデータ数を合わせて検定した．検定結果を**表 7** に示す．表 7 より，コードクローンの上下で欠陥発生率に有意差が見られたのは evolution のみであった．つまり，コードクローンの上下では欠陥発生率に差がないと言える．

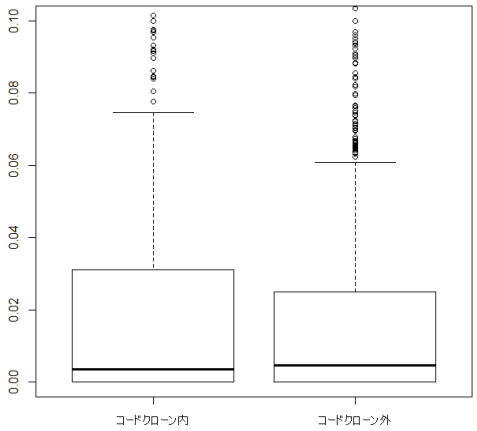

図 5 コードクローン内外における欠陥発生率の比較（gimp）
Fig. 5 Comparison of defect rate within code clone and the surrounding code fragments on gimp.

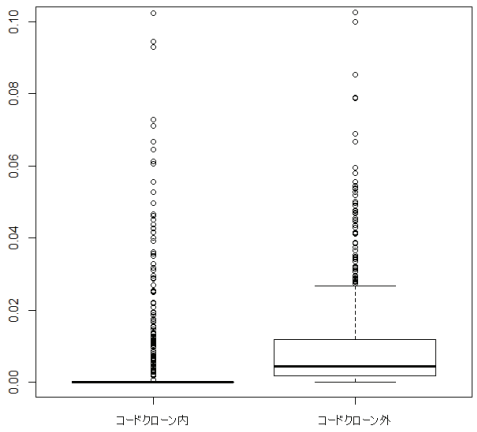

図 6 コードクローン内外における欠陥発生率の比較（evolution）
Fig. 6 Comparison of defect rate within code clone and the surrounding code fragments on evolution.

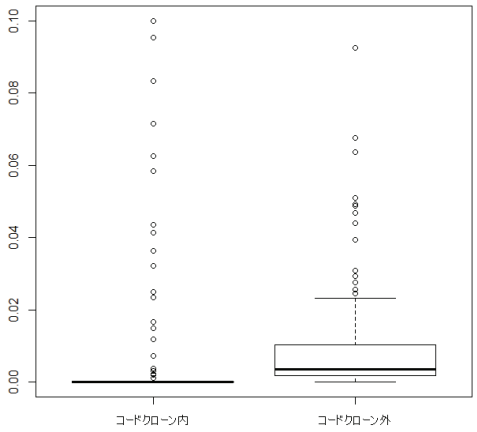

図 7 コードクローン内外における欠陥発生率の比較（nautilus）
Fig. 7 Comparison of defect rate within code clone and the surrounding code fragments on nautilus.

表 2　調査 1 の検定結果
Table 2　Result of investigation1.

	中央値（コードクローン内）	中央値（コードクローン外）	p 値（Wilcoxon の順位和検定）
gimp	0.003565	0.004619	p-value = 0.2783
evolution	0	0.003601	p-value < 2.2e-16
nautilus	0	0.004537	p-value < 2.2e-16

表 3　コードクローンの上側におけるコードクローンから近い範囲と遠い範囲の比較（1 対 1）
Table 3　Comparison between near fragment and far fragment (half-and-half).

	平均値（近）	平均値（遠）	p 値（t 検定）
gimp	0.02874	0.02745	1.778e-06
evolution	0.01742	0.01741	0.9643
nautilus	0.01079	0.008713	5.453e-08

表 4　コードクローンの上側におけるコードクローンから近い範囲と遠い範囲の比較（1 対 2）
Table 4　Comparison between near fragment and far fragment (one to two).

	平均値（近）	平均値（遠）	p 値（t 検定）
gimp	0.02870	0.02778	0.002098
evolution	0.01729	0.01748	0.1229
nautilus	0.010483333	0.009391	0.009355

表 5　コードクローンの下側におけるコードクローンから近い範囲と遠い範囲の比較（1 対 1）
Table 5　Comparison between near fragment and far fragment (half-and-half).

	平均値（近）	平均値（遠）	p 値（t 検定）
gimp	0.02839	0.02867	0.6363
evolution	0.01683	0.01668	0.4509
nautilus	0.01043	0.01049	0.9051

表 6　コードクローンの下側におけるコードクローンから近い範囲と遠い範囲の比較（1 対 2）
Table 6　Comparison between near fragment and far fragment (one to two).

	平均値（近）	平均値（遠）	p 値（t 検定）
gimp	0.02779	0.02893	0.1647
evolution	0.01694	0.01666	0.1882
nautilus	0.009567	0.01092	0.006858

表 7　コードクローンの上側と下側の差の検定
Table 7　Comparison between upper fragment and lower fragment.

	平均値（上側）	平均値（下側）	p 値（t 検定）
gimp	0.02876	0.02852	0.5375
evolution	0.01742	0.01676	3.459e-06
nautilus	0.01029	0.01046	0.6688

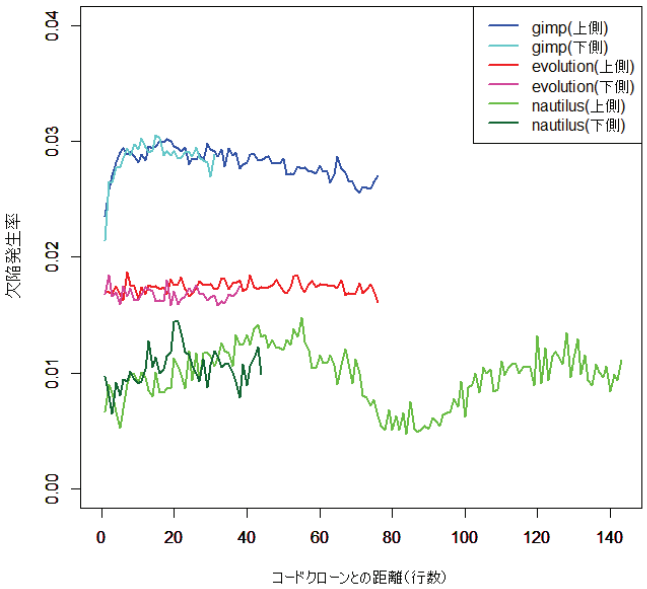

図 8　コードクローン外における欠陥発生率の推移
Fig. 8　Changes of defect rate within the surrounding code fragments.

5. 考察

5.1 結果の考察

調査 1 では，全てのプロジェクトでコードクローン外における欠陥発生率の中央値がコードクローン内における欠陥発生率の中央値よりも高かった．gimp については，コードクローン内外の中央値に統計的な有意差は見られなかったが，残り 2 つのプロジェクトでは有意差が見られた．したがって，コードクローン内外で欠陥発生率を比較すると，コードクローン外の方が欠陥発生率が高い傾向にあると言える．ゆえに，開発者がコードクローンに着目して欠陥を探す際は，コードクローン外にも欠陥が存在する可能性が十分にあるという前提で検査すべきと考えられる．調査 2 の gimp と nautilus に関する調査結果では，コードクローンの上側における欠陥発生率がコードクローンから遠ざかる（距離が大きくなる）ほど低下することが確認された．また，コードクローンの上側と下側で欠陥発生率に差があるか検定した結果，両者に差は見られなかった．したがって，開発者はコードクローンの上側と下側の両方に対して

欠陥を検査する必要があるが，コードクローンの上を検査する際は，コードクローンの近辺を優先的に検査すべきと考えられる．

5.2 妥当性への脅威

調査対象数の妥当性

本研究では 3 つの OSS を対象に調査を行ったが，全ての調査項目において 3 つの OSS プロジェクトに共通した傾向は確認されなかった．また，欠陥発生率を比較した結果，比較対象に有意差が見られたプロジェクトは複数存在したが，そのプロジェクトも調査項目によって異なっていた．そのため，より一般性のある知見を得るためには，さらに多くの OSS に対して調査を行う必要がある．

コードクローン検出ツールの妥当性

本調査では，コードクローンの検出に Jiang らの DECKARD を用いたが，コードクローンの検出手法は他にも多くの研究者によって提案されている [2][13][14]．それぞれ検出アルゴリズムが異なるため，たとえ検出の対象が同じであっても，手法によって検出するコードクローンの数や位置は異なると考えられる．ゆえに，他の手法やツールを用いて調査を行った場合，本研究においてコードクローン内と判断したコード片がコードクローン外となり調査結果が変わる恐れがある．しかし，DECKARD の検出手法では，一般的に用いられている他のいくつかの手法で検出できないコードクローンを検出することができる．それは，図 2 で示したコピーアンドペースト後にコード内容が修正されたコードクローンである．コードクローンの編集忘れが原因で欠陥が発生する場合があることからも，本研究におけるコードクローン検出には DECKARD が適していると考えられる．

6. まとめと今後の課題

本研究では，コードクローン内外の欠陥発生率およびコードクローン外における欠陥発生率の推移を 3 つの OSS に対して調査した．コードクローン内外の欠陥発生を比較したところ，2 つのプロジェクトでコードクローン外の方が欠陥発生率が高いことが確認された．また，コードクローンとの距離に基づきコードクローン外における欠陥発生の推移を調べた結果，2 つのプロジェクトでコードクローンの上側ではコードクローンから遠ざかるほど欠陥発生率が低下する傾向が見られた．また，コードクローンの上側と下側では欠陥発生率に差がないことも確認された．

今後の課題としては，より一般性のある知見を得るために，他の OSS を対象とした調査を行うことや他のコードクローン検出ツールを用いて再調査することが挙げられる．

謝辞

本研究は日本学術振興会科学研究費補助金（課題番号：26730036）の助成を受けている．

参考文献

[1] 肥後芳樹，吉田則裕: コードクローンを対象としたリファクタリング，コンピュータソフトウェア，Vol.28，No.4（2011），pp.43-56.
[2] Kamiya, T.,Kusumoto, S. and Inoue, K. : CCFinder : a Multilinguistic Token-Based Code Clone Detection System for Large Scale Source Code, IEEE Trans. Softw. Eng, Vol28, No.7, pp.654-670 (2002).
[3] Li, Z., Lu, S., Myagmar, S. and Zhou, Y.: CP-Miner: Finding Copy-Paste and Related Bugs in Large-Scale Software Code, IEEE Trans. Softw. Eng., Vol.32, No.3, pp.176-192 (2006).
[4] Jiang, L., Su, Z. and Chiu, E.: Context-Based Detection of Clone-Related Bugs, Proc of ESEC/FSE 2007, 2007, pp.147-156.
[5] Rahman, F., Bird, C. and Devanbu, P.: Clones: what is that smell?, Proc of MSR 2010, 2010, pp.72-81.
[6] Rahman, F., Bird, C. and Devanbu, P.: Clones: what is that smell?, Empir Software Eng, Vol.17, Issue 4-5, pp.503-530 (2012).
[7] Sajnani, H., Saini, V. and Lopes, C.: A Comparative Study of Bug Patterns in Java Cloned and Non-cloned Code, Proc of SCAM 2014, 2014, pp.21-30.
[8] FindBugs : FindBugs^{TM} Manual(online), 入手先 ⟨http://findbugs.sourceforge.net/manual/index.html⟩ (2014.10.10).
[9] Hovemeyer, D. and Pugh, W: Finding bugs is easy., SIGPLAN Not., 39(12):92-106, Dec. 2004.
[10] Śliwerski, J., Zimmermann, T and Zeller, Andreas: When Do Changes Induce Fixes?, Proc of MSR 2005, 2005, pp.24-28.
[11] Bachmann, A and Bernstein, A: Data Retrieval, Processing and Linking for Software Process Data Analysis, Technical Report No.IFI-2009.0003b, Department of Informatics University of Zurich.
[12] Jiang, L., Misherghi, G, Su, Z and Glondu, S: DECKARD:Scalable and Accurate Tree-Based Detection of Code Clones, Proc of ICSE 2007, 2007, pp.96-105.
[13] Baxter, I., Yahin, A., Moura, L., Annna, M and Bier, L: Clone Detection Using Abstract Syntax Trees, Proc. of ICSM 1998, 1998, pp.368-377.
[14] 村上寛明，堀田圭佑，肥後芳樹，井垣宏，楠本真二: Smith-Waterman アルゴリズムを利用したギャップを含むコードクローン検出，情報処理学会論文誌，Vol.55，No.2（2014），pp.981-993.